ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, IL 60439

ANL/MCS-TM-205

Nexus: Runtime Support for Task-Parallel Programming Languages

by

Ian Foster, Carl Kesselman, and Steven Tuecke*

Mathematics and Computer Science Division

Technical Memorandum No. 205

February 1995 (draft)

This work was supported by the Office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38, and by the National Science Foundation's Center for Research in Parallel Computation under Contract CCR-8809615.

*Dept. of Computer Science, California Institute of Technology, Pasadena, CA 91125.

Contents

Nexus: Runtime Support for Task-Parallel Programming Languages

Ian Foster Carl Kesselman Steven Tuecke

Abstract

A runtime system provides a parallel language compiler with an interface to the low-level facilities required to support interaction between concurrently executing program components. Nexus is a portable runtime system for task-parallel programming languages. Distinguishing features of Nexus include its support for multiple threads of control, dynamic processor acquisition, dynamic address space creation, a global memory model via interprocessor references, and asynchronous events. In addition, it supports heterogeneity at multiple levels, allowing a single computation to utilize different programming languages, executables, processors, and network protocols. Nexus is currently being used as a compiler target for two task-parallel languages: Fortran M and Compositional C++. In this paper, we present the Nexus design, outline techniques used to implement Nexus on parallel computers, show how it is used in compilers, and compare its performance with that of another runtime system.

1 Introduction

Compilers for parallel languages rely on the existence of a runtime system. The runtime system defines the compiler's view of a parallel computer: how computational resources are allocated and controlled and how parallel components of a program interact, communicate and synchronize with one another.

Most existing runtime systems support the single-program, multiple-data (SPMD) programming model used to implement data-parallel languages such as High Performance Fortran (HPF) [10], Fortran-D [18], Vienna Fortran [6], and pC++ [17]. In this model, each processor in a parallel computer executes a copy of the same program. Processors exchange data and synchronize with each other through calls to the runtime library, which typically is designed to optimize collective operations in which all processors communicate at the same time, in a structured fashion. A major research goal in this area is to identify common runtime systems that can be shared by a variety of SPMD systems.

Task-parallel computations extend the SPMD programming paradigm by allowing unrelated activities to take place concurrently. The need for task parallelism arises in time-dependent problems such as discrete-event simulation, in irregular problems such as sparse matrix problems, and in multidisciplinary simulations coupling multiple, possibly data-parallel, computations. Task-parallel programs may dynamically create multiple, potentially unrelated, threads of control. Communication and synchronization are between threads, rather than processors, and can occur asynchronously among any subset of threads and at any point in time. A compiler often has little global information about a task-parallel computation, so there are few opportunities for exploiting optimized collective operations.

In the long term, it may prove possible to design runtime systems that support both task-parallel and SPMD computations efficiently. Before such integrated systems can be designed, however, it is important to identify the runtime requirements of task-parallel languages, and to develop efficient and portable runtime systems that meet these requirements. In order to achieve these goals, different task-parallel language projects need to collaborate to identify common runtime requirements and to experiment with the use of common runtime support. The Nexus project represents a first step in this direction.

The design of Nexus is shaped both by the requirements of task-parallel computations and by a desire to support the use of heterogeneous environments, in which heterogeneous collections of computers may be connected by heterogeneous networks. Other design goals include efficiency, portability across diverse systems, and support for interoperability of different compilers. It is not yet clear to what extent these various goals can be satisfied in a single runtime system: in particular, the need for efficiency may conflict with the need for portability and heterogeneity. Later in this paper, we present some preliminary performance results that address this question.

As we describe in this paper, Nexus is already in use as a compiler target for two task-parallel languages: Fortran M (FM) and Compositional C++ (CC++). Our initial experiences have been gratifying in that the resulting compilers are considerably simpler than earlier prototypes that did not use Nexus services. Nevertheless, further work is required to determine whether the Nexus design incorporates features that will be useful for a wide range of task-parallel computations. One of our goals in writing this paper is to encourage discussion of this topic.

2 Existing Runtime Systems

We first review some existing runtime systems. We focus on systems designed for distributed-memory computers, motivated by the prevalence of this architecture among large, scalable parallel computers.

At the lowest level, parallel runtime systems must support data transfer between processors and synchronization on the availability of data. The mechanisms most

commonly used for these purposes are *send* and *receive*. Send calls are usually addressed to a processing node, which in a data-parallel program will be executing the same program as the sending node. Consequently, it is straightforward for a compiler to place a corresponding receive in the generated code. The send/receive model is supported by a variety of machine-specific and portable communication libraries, including NX, p4, PVM, and MPI [3, 11, 8]. These are designed for programmer use and are not necessarily good compiler targets. In particular, the focus on process-based rather than thread-based communication causes difficulty for task-parallel languages.

Two representative runtime systems layered on top of a send/receive model are the HPF runtime of Bozkus et al. [1] and CHAOS [21]. Both support an SPMD programming model. In Bozkus et al.'s runtime, the focus is on providing efficient support for collective operations on distributed arrays. Services include rotation of a matrix by row and column and broadcast along specific dimensions. A global Fortran namespace is supported by routines that map between indices for local sections of arrays and the global indices associated with that data. CHAOS supports irregular mesh computations in data-parallel languages. A technique called runtime compilation is used to compute optimized communication schedules at runtime, which are then executed in an SPMD fashion.

While apparently effective for data-parallel computation, the send/receive model poses difficulties for task-parallel systems. Because communication is between threads, not nodes, and can take place asynchronously, it can be difficult for a compiler to place receive operations. In addition, few existing send/receive libraries are thread safe.

A promising alternative model is *active messages* [22]. Here, a sender specifies the data that is to be transferred and the address of a compiler-generated active message handler that will process the data. When the data arrives at the destination processor, an interrupt is generated and the specified handler is executed as the interrupt handler. However, while active messages allow for asynchronous transfer, limitations on their semantics (enforced by a need to run in interrupt service routines) still restrict their use to data-parallel programs. As we will see, the *remote service request* used in Nexus behaves like an active message handler, but removes the restrictions that prevent its effective use in task-parallel programs.

The runtime systems discussed so far are subroutine libraries. An alternative approach is to define an abstract machine that defines the runtime environment. An abstract machine provides an instruction set tailored to compilation of the parallel language; the compiler translates programs into this instruction set. Examples include the Program Composition Machine (PCM) [16] and the Threaded Abstract Machine (TAM) [23]. PCM was designed as a compilation target for the task-parallel language PCN. It provides task creation, memory management via distributed garbage collection, synchronization via data-flow variables, and data transfer functions. TAM was designed to support the compilation of the data-flow language ID 90. The instruction set provides efficient support for the dynamic cre-

ation of multiple threads of control, locality of reference via hierarchal scheduling mechanisms, and efficient synchronization via data-flow variables.

3 Nexus Design and Implementation

Before describing the Nexus interface and implementation, we review the requirements and assumptions that motivated the Nexus design.

Nexus is intended as a *general-purpose runtime system* for task-parallel languages. While it currently contains no specialized support for data parallelism, data-parallel languages such as pC++and HPF can in principle also use it as a runtime layer. Nexus is designed specifically as a *compiler target*, not as a library for use by application programmers. Consequently, the design favors efficiency over ease of use.

We believe that the future of parallel computing lies in *heterogeneous environments* in which diverse networks and communications protocols interconnect PCs, workstations, small shared-memory machines, and large-scale parallel computers. We also expect *heterogeneous applications* combining different programming languages, programming paradigms, and algorithms to become widespread.

Nexus abstractions need to be close to the hardware, in order to provide *efficiency* on machines that provide appropriate low-level support. Operations that occur frequently in task-parallel computations, such as thread creation, thread scheduling, and communication, need to be particularly fast. At the same time, Nexus abstractions must be easily layered on top of existing runtime mechanisms, so as to provide *portability* to machines that do not support Nexus abstractions directly. Communication mechanisms that were considered in designing Nexus include message passing, shared memory, distributed shared memory, and message-driven computation.

Finally, Nexus is intended to be a *lingua franca* for compilers, promoting reuse of code between compiler implementation as well as *interoperability* between code generated by different compilers.

Important issues purposefully not addressed in the initial design include reliability and fault tolerance, real-time issues, global resource allocation, replication, data and code migration, and scheduling policies. We expect to examine these issues in future research.

3.1 Core Abstractions

The Nexus interface is organized around five basic abstractions: nodes, contexts, threads, global pointers, and remote service requests. The associated services provide direct support for light-weight threading, address space management, communication, and synchronization [14]. A computation consists of a set of *threads*, each executing in an address space called a *context*. An individual thread executes a sequential program, which may read and write data shared with other threads

executing in the same context. It can also generate asynchronous *remote service requests*, which invoke procedures in other contexts.

Nodes. The most basic abstraction in Nexus is that of a *node*. A node represents a physical processing resource. Consequently, the set of nodes allocated by a program determines the total processing power available to that computation. When a program using Nexus starts, an initial set of nodes is created; nodes can also be added or released dynamically. Programs do not execute directly on a node. Rather, as we will discuss below, computation takes place in a context, and it is the context that is mapped to a node.

Nexus provides a set of routines to create nodes on named computational resources, such as a symmetric shared-memory multiprocessor or a processor in a distributed-memory computer. A node specifies only a computational resource and does not imply any specific communication medium or protocol. This naming strategy is implementation dependent; however, a node can be manipulated in an implementation-independent manner once created.

Contexts. Computation takes place within an object called a *context*. Each context relates an executable code and one or more data segments to a node. Many contexts can be mapped onto a single node. Contexts cannot be migrated between nodes once created.

Contexts are created and destroyed dynamically. We anticipate context creation occurring frequently: perhaps every several thousand instructions. Consequently, context creation should be inexpensive: certainly less expensive then process creation under Unix. This is feasible because unlike Unix processes, contexts do not guarantee protection. We note that the behavior of concurrent I/O operations within contexts is currently undefined.

Compiler-defined initialization code is executed automatically by Nexus when a context is created. Once initialization is complete, a context is inactive until a thread is created by an explicit remote service request to that context. The creation operation is synchronized to ensure that a context is not used before it is completely initialized. The separation of context creation and code execution is unique to Nexus and is a direct consequence of the requirements of task parallelism. All threads of control in a context are equivalent, and all computation is created asynchronously.

Threads. Computation takes place in one or more *threads* of control. A thread of control must be created within a context. Nexus distinguishes between two types of thread creation: within the same context as the currently executing thread and in a different context from the currently executing thread. We discuss thread creation between contexts below.

Nexus provides a routine for creating threads within the context of the currently executing thread. The number of threads that can be created within a context is limited only by the resources available. The thread routines in Nexus are modeled

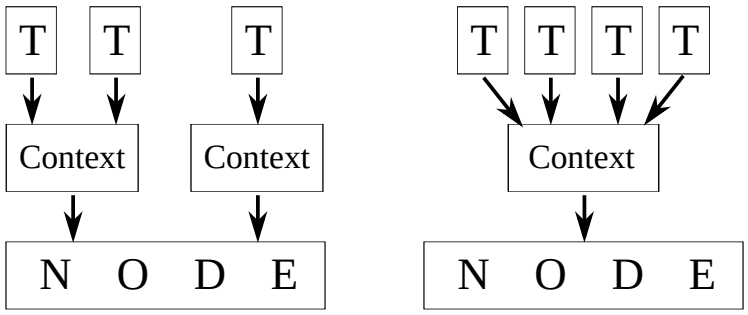

Figure 1: *Nodes, Contexts, and Threads*

after a subset of the POSIX thread specification [20]. The operations supported include thread creation, termination, and yielding the current thread. Mutexes and condition variables are also provided for synchronization between threads within a context.

Basing Nexus on POSIX threads was a pragmatic choice: because most vendors support POSIX threads (or something similar), it allows Nexus to be implemented using vendor-supplied thread libraries. The drawback to this approach is that POSIX was designed as an application program interface, with features such as real-time scheduling support that may add overhead for parallel systems. A lower-level interface designed specifically as a compiler target would most likely result in better performance [2, 9] and will be investigated in future research.

To summarize, the mapping of computation to physical processors is determined by both the mapping of threads to contexts and the mapping of contexts to nodes. The relationship between nodes, contexts, and threads is illustrated in Fig. 1.

Global Pointers. Nexus provides the compiler with a global namespace, by allowing a global name to be created for any address within a context. This name is called a *global pointer*. A global pointer can be moved between contexts, thus providing for a movable intercontext reference. Global pointers are used in conjunction with remote service requests to cause actions to take place on a different context. The use of global pointers was motivated by the following considerations.

- While the data-parallel programming model naturally associates communication with the section of code that generates or consumes data, task-parallel programs need to associate the communication with a specific data structure or a specific piece of code. A global namespace facilitates this.

- Dynamic behaviors are the rule in task-parallel computation. References to data structures need to be passed between contexts.

- Data structures other than arrays need to be supported. A general global pointer mechanism facilitates construction of complex, distributed data structures.

- Distributed-memory computers are beginning to provide direct hardware support for a global shared namespace. We wanted to reflect this trend in Nexus.

Global pointers can be used to implement data structures other than C pointers. For example, the FM compiler uses them to implement channels.

Remote Service Requests. A thread can request that an action be performed in a remote context by issuing a *remote service request*. A remote service request results in the execution of a special function, called a *handler*, in the context pointed to by a global pointer. The handler is invoked asynchronously in that context; no action, such as executing a receive, needs to take place in the context in order for the handler to execute. A remote service request is not a remote procedure call, because there is no acknowledgement or return value from the call, and the thread that initiated the request does not block.

Remote service requests are similar in some respects to active messages [22]. They also differ in significant ways, however. Because active message handlers are designed to execute within an interrupt handler, there are restrictions on the ways in which they can modify the environment of a node. For example, they cannot call memory allocation routines. While these restrictions do not hinder the use of active messages for data transfer, they limit their utility as a mechanism for creating general threads of control. In contrast, remote service requests are more expensive but less restrictive. In particular, they can create threads of control, and two or more handlers can execute concurrently.

During a remote service request, data can be transferred between contexts by the use of a *buffer*. Data is inserted into a buffer and removed from a buffer through the use of packing and unpacking functions similar to those found in PVM and MPI [8, 11]. Invoking a remote service request is a three-step process:

1. The remote service request is initialized by providing a global pointer to an address in the destination context and the identifier for the handler in the remote context. A buffer is returned from the initialization operation.

2. Data to be passed to the remote handler is placed into the buffer. The buffer uses the global pointer provided at initialization to determine if any data conversion or encoding is required.

3. The remote service request is performed. In performing the request, Nexus uses the global pointer provided at initialization to determine what communication protocols can be used to communicate with the node on which the context resides.

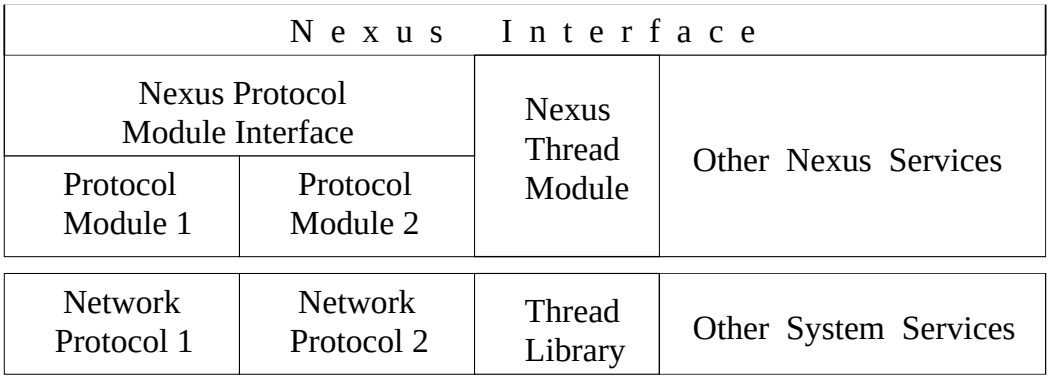

Figure 2: *Structure of Nexus Implementation*

The handler is invoked in the destination context with the local address component of the global pointer and the message buffer as arguments. In the most general form of remote service request, the handler runs in a new thread. However, a compiler can also specify that a handler is to execute in a preallocated thread if it knows that that handler will terminate without suspending. This avoids the need to allocate a new thread; in addition, if a parallel computer system allows handlers to read directly from the message interface, it avoids the copying to an intermediate buffer that would otherwise be necessary for thread-safe execution. As an example, a handler that implements the get and put operations found in Split-C [7] can take advantage of this optimization.

3.2 Implementation

In our description of the Nexus implementation, we focus on the techniques used to support execution in heterogeneous environments: in particular, to support multiple communication protocols. As an example of why this is important, the IBM SP1 at Argonne (a representative modern parallel computer, with multifunctional nodes) currently provides five communication protocols, any combination of which may be used in a particular application: shared memory between processes on the same node, MPI over the messaging fabric, Fiber Channel from compute nodes to I/O nodes, HIPPI between I/O nodes, and TCP to other computers. In the future, Asynchronous Transfer Mode (ATM) connections will also be incorporated. While TCP can be used on many of these networks, it is often not the most efficient protocol.

In order to support heterogeneity, the Nexus implementation encapsulates thread and communication functions in thread and protocol modules, respectively, that implement a standard interface to low-level mechanisms (Fig. 2). Current thread modules include POSIX threads, DCE threads, C threads, and Solaris threads. Cur-

rent protocol modules include local (intracontext) communication, TCP socket, and Intel NX message-passing. Protocol modules for MPI, PVM, SVR4 shared memory, Fiber Channel, IBM's EUI message-passing library, AAL-5 (ATM Adaptation Layer 5) for Asynchronous Transfer Mode (ATM), and the Cray T3D's get and put operations are planned or under development.

More than one communication mechanism can be used within a single program. For example, a context A might communicate with contexts B and C using two different communication mechanisms if B and C are located on different nodes. This functionality is supported as follows. When a protocol module is initialized, it creates a table containing the functions that implement the low-level interface and a small descriptor that specifies how this protocol is to be used. (Protocol descriptors are small objects: typically 4-5 words, depending on the protocol.) When a global pointer is created in a context, a list of descriptors for the protocols supported by this context is attached to the global pointer. The protocol descriptor list is part of the global pointer and is passed with the global pointer whenever it is transferred between contexts. A recipient of a global pointer can compare this protocol list with its local protocols to determine the best protocol to use when communicating on that global pointer.

Although some existing message-passing systems support limited network heterogeneity, none do so with the same generality. For example, PVM3 allows processors in a parallel computer to communicate with external computers by sending messages to the `pvmd` daemon process which acts as a message forwarder [8]. However, this approach is not optimal on machines such as the IBM SP1 and the Intel Paragon, whose nodes are able to support TCP directly, and it limits PVM programs to using just one protocol in addition to TCP. P4 has several special multiprotocol implementations, such as a version for the Paragon that allows the nodes to use both NX and TCP [3]. But it does not allow arbitrary mixing of protocols.

4 Nexus as a Compiler Target

Nexus is currently being used as the runtime system for two different programming languages: CC++and FM. Although both languages provide a task-parallel programming model, they have very different characteristics. The compilers use Nexus in two different ways, translating some language constructs directly to Nexus calls and for others generating calls to specialized FM or CC++runtime libraries implemented using Nexus services.

In the following, we give a brief overview of CC++and FM and the ways in which they use Nexus services. Rather than present a comprehensive discussion of compilation strategies, our goal is to present representative examples that illustrate the correspondence between Nexus features and the runtime requirements of these languages.

4.1 Compiling CC++Using Nexus

CC++ [4, 5] is a general-purpose parallel programming comprising all of C++plus six new keywords. The CC++parallel constructs are intended to support the development of parallel class libraries implementing a wide range of different parallel programming styles, for example, synchronous virtual channels, actors, data flow, and concurrent aggregates.

Parallel Control Structures. CC++provides three parallel control structures: the parallel block, the parallel loop, and the spawn statement. These structures create new threads of control which have complete access to the environment in which they execute; they map naturally to Nexus threads.

Parallel blocks and loops introduce structured parallelism: they do not terminate until all threads of control that they have created terminate. Because Nexus does not provide any parent/child relationship between threads, the CC++compiler must place a barrier after each parallel block or loop. These barriers are implemented with Nexus condition variables.

Synchronization Structures. CC++provides two mechanisms for controlling the interaction of parallel threads of control: synchronization (sync) variables and atomic functions. Any data type can be made into a sync variable by modifying its type with the keyword `sync`. Sync variables are single assignment variables. Attempting to read an unassigned sync variable causes the reader to block; attempting to write to a sync variable more than once is an error. Assignment to a sync variable wakes any threads suspended on that variable. A sync variable is implemented as a data structure containing the variable's value, a flag to indicate if the variable has been initialized and a Nexus condition variable. Threads waiting for the value of a sync variable block on the condition variable.

Atomic functions provide a means for controlling the scheduling of threads. An atomic function is like a monitor [19]. Within an instance of a given C++class, only one atomic function is allowed to execute at a time. Atomic functions are implemented by requiring that they obtain a Nexus mutex prior to executing the function body.

Managing Processing Resources. CC++introduces a structure called a *processor object*. Like other C++objects, a processor object has a type declared by a class definition, encapsulates functions and data, and can be dynamically created and destroyed. It is distinguished from other objects by the type of resources from which it is allocated. A normal object is allocated from an address space, while the resources for a processor object are allocated from a "process space". Each instance of a processor object contains an address space from which regular objects can be allocated. Thus a CC++computation consists of a collection of processor objects, where each processor object contains a collection of normal C++objects.

Interprocessor object references are allowed but must be explicitly declared to be `global`. Global pointers provide CC++with both a global name space and a two-level locality model that can be manipulated directly by a program. A global pointer can be dereferenced like any other C++pointer. However, dereferencing a global pointer causes an operation to take place in the processor object referenced by that global pointer.

A Nexus context is created for each processor object created by a CC++program; the compiler also ensures that a Nexus node exists prior to creating a processor object. CC++global pointers are mapped directly into Nexus global pointers. An operation that results in dereferencing a global pointer is compiled to a remote service request. The handler for this request is specific to the data type of the global pointer and is generated by the compiler. The function of the handler is to perform the requested operation on the object referenced by the global pointer.

Data access and function call through global pointers are synchronous: the caller must wait until the operations complete and any return values have be obtained. Because Nexus remote service requests are asynchronous and unidirectional, a handler responsible for processing a remote operation must notify the initiator when that operation has completed. This notification is accomplished by issuing a remote service request back to the context from which the operation was invoked. The thread requesting the remote operation waits for completion by blocking on a Nexus condition variable. By including a global pointer to that condition variable as part of the data included in the initial remote service request, the return remote service request can signal on the condition variable, notifying the initial thread that the remote operation has completed. Because remote service requests are unidirectional, the CC++compiler can detect when a return value is not required and optimize out the return remote service request.

4.2 Compiling FM Using Nexus

FM [13, 15] is a small set of extensions to Fortran 77 for task-parallel programming. FM is designed to support both the modular construction of large parallel programs and the development of libraries implementing other programming paradigms. For example, in a joint project with Syracuse, such a library has been used to integrate HPF programs into a task-parallel framework [12].

FM programs can dynamically create and destroy *processes*, single-reader/single-writer *channels*, and multiple-writer, single-reader *mergers*. Processes can encapsulate state (common data) and communicate by sending and receiving messages on channels and mergers; references to channels, called *ports*, can be transferred between processes in messages. FM also provides constructs for mapping processes to processors.

Processes FM processes are created by process block and process do-loop constructs. These have similar semantics to CC++parallel blocks and loops and are

implemented in the same fashion, by using a compiler-generated barrier.

Arguments passed to a process are copied in on call and back on return. A process is compiled into two Nexus handlers: a process invocation handler which invokes the subroutine in a new thread, extracting arguments from the buffer; and a process return handler, called on process completion to return arguments to the calling process block and to update the barrier.

By default, an FM process is implemented as a thread executing in a dedicated Nexus context, with the context's data segments used to hold process state. This context must be allocated by the FM compiler prior to creating the thread, and deallocated upon process termination. As an optimization, processes without state can be implemented as threads in a preexisting context containing the code for that process. This optimization can reduce process creation costs and, in some systems, scheduling costs, and is important for fine-grained applications.

Channels and Mergers. A channel is a typed, first-in/first-out message queue with a single sender and a single receiver; the merger is similar but allows for multiple senders. A restricted global address space is provided by outport and inport variables, which can contain references to the sending and receiving ends, respectively, of channels and mergers. Ports can be passed as arguments when processes are created, or can be transferred between processes in messages.

A channel is implemented as a message queue data structure maintained in the context of the receiving process; an outport is implemented as a data structure containing a Nexus global pointer to the channel data structure. A send operation is compiled to code which packs the message data into a buffer and invokes a remote service request to a compiler-generated handler which enqueues the message onto the channel. A receive operation is compiled to code which unpacks a pending message into variables or suspends on a condition variable in the channel data structure if no messages are pending.

One of the more complex aspects of FM implementation is port migration. For efficiency reasons, we maintain the invariant that a channel data structure is located in the same context as its inport; hence, migration of a port can require the updating of a number of distributed data structures. In earlier versions of the FM compiler, these protocols were implemented in terms of send and receive calls; however, the resulting code was both complex and hard to verify. The asynchronous nature of the Nexus remote service request has greatly simplified both implementation and validation.

Process Mapping. FM constructs allow the programmer to control process placement by specifying mappings of processes to *virtual computers*: arrays of virtual processors. The mapping of virtual to physical processors is specified at program startup. The programmer can also define submachines to indicate that a subcomputation should execute in a subset of available resources.

A virtual processor array is implemented as an array of pointers to Nexus node structures. Mapping a process call to a virtual processor involves first looking up the correct node in the virtual processor array and then creating the process on that node (in a new or existing context). Creating a submachine causes a new virtual processor array, based on the existing one, to be created.

4.3 Interoperability between FM and CC++

Because CC++and FM are both implemented using Nexus facilities, parallel structures in the two languages can interact. For example, an FM program can invoke a CC++program, specifying the contexts in which it is to execute and passing as arguments an array of Nexus global pointers representing the inports or outports of channels. The CC++program can then send or receive functions on these global pointers to transfer data between contexts executing FM code and contexts executing CC++code.

5 Performance Studies

In this section, we present results of some preliminary Nexus performance studies. We note that the thrust of our development effort to date has been to provide a correct implementation of Nexus. No tuning or optimization work has been done at all. In addition, the operating system features used to implement Nexus are completely generic: we have not exploited even the simplest of operating system features, such as nonblocking I/O. Consequently, the results reported here should be viewed as suggestive of Nexus performance only, and are in no way conclusive.

The experiments that we describe are designed to show the cost of the Nexus communication abstraction as compared to traditional send and receive. Because Nexus-style communication is not supported on current machines, Nexus is implemented with send and receive. Thus, Nexus operations will have overhead compared to using send and receive. Our objective is to quantify this overhead. We note that support for Nexus can be build directly into the system software for a machine, in which case Nexus performance could meet or even exceed the performance of a traditional process-oriented send and receive based system. (We have started a development effort with the IBM T.J. Watson Research Center to explore this possibility.)

The experiments reported here compare the performance of a CC++program compiled to use Nexus and a similar C++program using PVM [8] for communication. The CC++program uses a function call through a CC++global pointer to transfer an array of double-precision floating-point numbers between two processor objects (Nexus contexts). We measure the cost both with remote thread creation and when a preallocated thread is used to execute the remote service request. The PVM program uses send and receive to transfer the array. Both systems are compiled with `-O3` using the Sun unbundled C and C++compilers; neither performs data conversion.

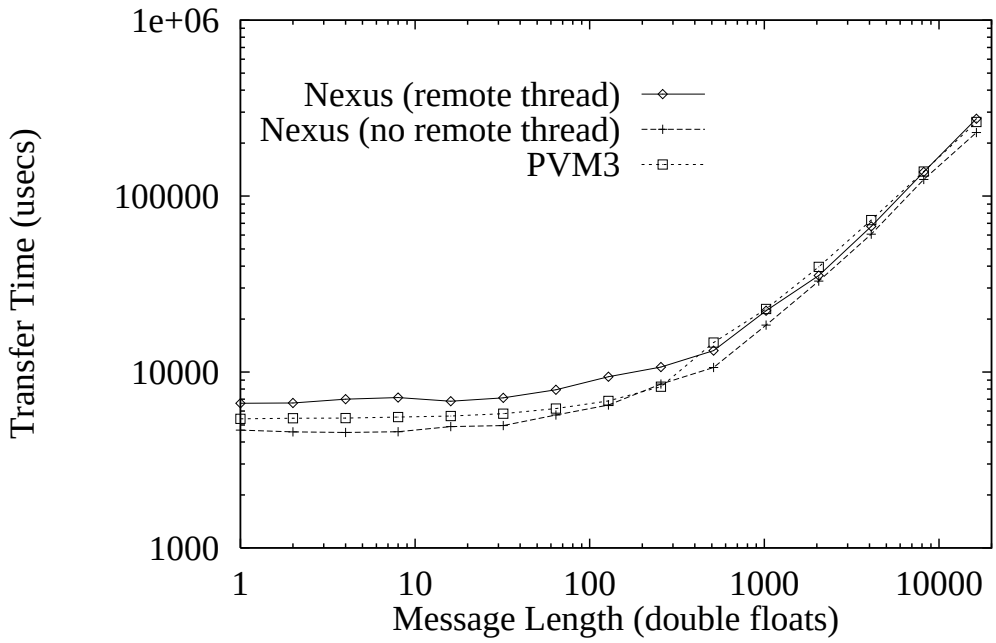

Figure 3: *Round-trip time as a function of message size between two Sun 10 workstations under Solaris 2.3 using an unloaded Ethernet.*

In both cases the data starts and finishes in a user-defined array. This array is circulated between the two endpoints repeatedly until the accumulated execution time is sufficient to measure accurately. Execution time is measured for a range of array sizes. The results of these experiments are summarized in Fig. 3.

We see that despite its lack of optimization, Nexus is competitive with PVM. Execution times are consistently lower by about 15 per cent when remote service requests are executed in a preallocated thread; this indicates that both latency and per-word transfer costs are lower. Not surprisingly, execution times are higher when a thread is created dynamically: by about 40 per cent for small messages and 10 to 20 per cent for larger messages.

6 Summary and Future Work

Nexus is a runtime system for compilers of task-parallel programming languages. It provides an integrated treatment of multithreading, address space management, communication, and synchronization and supports heterogeneity in architectures and communication protocols.

Nexus is operational on networks of Unix workstations communicating over TCP/IP networks, the IBM SP1, and the Intel Paragon using NX; it is being ported to other platforms and communication protocols. Nexus has been used to implement two very different task-parallel programming languages: CC++and Fortran M. In both cases, the experience with the basic abstractions has been positive: the overall complexity of both compilers was reduced considerably compared to earlier prototypes that did not use Nexus facilities. In addition, we have been able to reuse code and have laid the foundations for interoperability between the two compilers. The preliminary performance studies reported in this paper suggest that Nexus facilities are competitive with other runtime systems.

These preliminary experiences suggest that Nexus is already useful as a tool for implementing task-parallel programming languages, as a framework for studying the interactions between task and data parallelism, and as a vehicle for studying parallelism on heterogeneous computer systems and networks. In future work, we plan to extend the basic Nexus design to incorporate new capabilities, including I/O operations and support for data-parallel computations. In the latter area, our objective is not primarily to support purely data-parallel computations, but rather to support programs which combine task and data parallelism. This work is being pursued jointly with other members of the POrtable RunTime System consortium (PORTS), a working group including universities, government laboratories, and industry.

We also plan to conduct more detailed investigations of the performance consequences of Nexus interface and implementation design decisions. For example, we want to understand the cost of compile-time management of storage associated with communication, and determine whether our compilers can provide the communication layer with additional information regarding data structures to facil-

itate optimization. We are also interested in the performance implications of the POSIX-based thread interface, and the potential benefits of lower-level interfaces or lighter-weight threads. Finally, we wish to investigate the locality properties of compiler-generated Nexus code, to determine whether a hierarchical scheduling mechanism such as found in TAM [23] can improve performance.

Acknowledgments

We are grateful to Bob Olson and James Patton for their considerable input to the Nexus design and implementation. The FM runtime support was designed and implemented by Robert Olson, and the NX protocol module by Tal Lancaster.

References

[1] Z. Bozkus, Alok Choudhary, Geoffrey C. Fox, T. Haupt, and S. Ranka. Fortran 90D/HPF compiler for distributed memory MIMD computers: Design, implementation, and performance results. In *Proc. Supercomputing '93*. IEEE, November 1993.

[2] Peter Buhr and R. Stroobosscher. The μsystem: Providing light-weight concurrency on shared-memory multiprocessor systems running Unix. *Software Practice and Experience*, pages 929–964, September 1990.

[3] R. Butler and E. Lusk. Monitors, message, and clusters: The p4 parallel programming system. *Parallel Computing (to appear)*, 1994.

[4] K. Mani Chandy and Carl Kesselman. CC++: A declarative concurrent object oriented programming notation. In *Research Directions in Object Oriented Programming*. MIT Press, 1993.

[5] K. Mani Chandy and Carl Kesselman. Compositional C++: Compositional parallel programming. In *Proc. Fifth Int'l Workshop on Parallel Languages and Compilers*. Springer-Verlag, 1993.

[6] Barbra Chapman, Piyush Mehrotra, and Hans Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, 1992.

[7] David Culler et al. Parallel programming in Split-C. In *Proc. Supercomputing '93*. ACM, 1993.

[8] J. Dongarra, G. Geist, R. Manchek, and V. Sunderam. Integrated PVM framework supports heterogeneous network computing. In *Computers in Physics*, April 1993.

[9] D. Engler, G. Andrews, and D. Lowenthal. Filaments: Efficient support for fine-grained parallelism. Technical Report 93-13, Dept. of Computer Science, U. Arizona, Tuscon, Ariz., 1993.

[10] High Performance Fortran Forum. High performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, Houston, Texas, January 1993.

[11] Message Passing Interface Forum. Document for a standard messge-passing interface, March 1994. (available from netlib).

[12] I. Foster, B. Avalani, A. Choudhary, and M. Xu. A compilation system that integrates High Performance Fortran and Fortran M. In *Proc. 1994 Scalable High Performance Computing Conf.* IEEE, 1994. to appear.

[13] Ian Foster and K. Mani Chandy. Fortran M: A language for modular parallel programming. *J. Parallel and Distributed Computing*, 1994. to appear.

[14] Ian Foster, Carl Kesselman, Robert Olson, and Steve Tuecke. Nexus: An interoperability toolkit for parallel and distributed computer systems. Technical Report ANL/MCS-TM-189, Argonne National Laboratory, 1994.

[15] Ian Foster, Bob Olson, and Steve Tuecke. Programming in Fortran M. Technical Report ANL-93/26, Argonne National Laboratory, 1993.

[16] Ian Foster and Stephen Taylor. A compiler approach to scalable concurrent program design. *ACM TOPLAS*, 1994. to appear.

[17] Dennis Gannon et al. Implementing a parallel C++ runtime system for scalable parallel systems. In *Proc. Supercomputing '93*, November 1993.

[18] Seema Hiranandani, Ken Kenedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD distributed memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[19] C.A.R Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[20] IEEE. Threads extension for portable operating systems (draft 6), February 1992.

[21] Ravi Ponnusamy, Joel Saltz, and Alok Choudhary. Runtime-compilation techniques for data partitioning and communication schedule reuse. Computer Science Technical Report CS-TR-3055, University of Maryland, 1993.

[22] Thorsten von Eicken, David Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. In *Proc. 19th Int'l Symposium on Computer Architecture*, May 1992.

[23] Thorsten von Eicken, David Culler, Seth Copen Goldstein, and Klaus Erik Schauser. TAM — a compiler controlled threaded abstract machine. *J. Parallel and Distributed Computing*, 1992.